Semiannual Technical Summary

Graphics

30 November 1967

Prepared for the Advanced Research Projects Agency
under Electronic Systems Division Contract AF 19(628)-5167 by

Lincoln Laboratory
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Lexington, Massachusetts

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LINCOLN LABORATORY

GRAPHICS

SEMIANNUAL TECHNICAL SUMMARY REPORT
TO THE
ADVANCED RESEARCH PROJECTS AGENCY

1 JUNE – 30 NOVEMBER 1967

ISSUED 4 JANUARY 1968

LEXINGTON MASSACHUSETTS

SUMMARY

The APEX display executive has been in regular operation and experience with new display hardware and a remote display console has been helpful in pinpointing needed changes. The APEX interrupt executive has been completed and will be evaluated during the next quarter. A new APEX scheduling algorithm has been implemented in order to improve the system response to users. The 338 remote console is operational in Washington, D. C., and is being used on a routine basis.

The compiler-compiler system VITAL is in regular use. The programming language LEAP, implemented using VITAL, has been used for programming constraint problems, a portion of the new VITAL system, integrated circuit mask layout, and two debugging packages. These latter present their data graphically to the user. The design of an improved VITAL system is under way.

The earth display program created earlier in the year has been used on actual data obtained from LES-4. A new approach for solving constrained systems is under development which uses a geometrical model. Several trial applications of this method have been programmed using LEAP. A special compiler and language for testing integrated circuits have been developed.

A theory of generalized superposition has been applied to waveform processing. Programs have been developed for scanning, processing, and photographing pictures which are subjected to this new kind of filtering. Programs for integrated circuit mask layout are under development which will allow designers to sketch in mask components and rearrange them in search for a satisfactory layout. Photographs can then be taken of the resulting masks.

Accepted for the Air Force
Franklin C. Hudson
Chief, Lincoln Laboratory Office

CONTENTS

GLOSSARY

ALGOL	High-level algebraic problem-solving language
APEX	TX-2 time-sharing executive
CORAL	A Lincoln Laboratory developed list processing language
CPU	Central processing unit
LABGOL	An ALGOL-like language constructed for the TX-2 using VITAL
LEAP	Extension of LABGOL, including associative language
LISP	A list-processing language
VITAL	A compiler-compiler system

GRAPHICS

I. SYSTEM PROGRAMMING

A. CRT Displays in APEX System

1. Introduction

The CRT display of graphic information from programs running under the APEX system is provided by a display section of the APEX executive. This display executive provides a means for building and manipulating a structured display file and handles the refresh buffering and necessary buffer storage management. The display generator is driven from the core buffered display file by a data channel with limited command capability. The paging hardware is used to transform the channel data reference addresses so that the display file need not occupy physically contiguous core pages. For both the display executive itself and its display buffers, the standard core page allocation mechanisms of APEX are used. The buffer pages are frozen in core so that a user's display is maintained even when his program and data have been swapped out to the drum. The display executive services the display at each of the five on-line consoles and also provides a network display capability for remote console use.

The display at any console is controlled by calls on the display executive from console user programs. The calls to the executive are of four basic kinds: on-off, item, group, and report. The on-off call initializes a user's display and can save and restore display files as user files on the drum. The item and group calls reflect the conceptual hierarchy of the display structure. An item is a collection of points, lines, curves, or characters with a single identifying name; it is the smallest display unit, but may contain many display components and create a complex picture. A group is an identified collection of items and may also contain a "use" of another group. This provides the structural subroutining now usual in display files. A reserved group, Group Zero, is the actual display and anything in its structure will appear on the scope.

Mode options on the item and group calls control light-pen priority as well as blank-unblank for each display part. The display file maintained by the executive for each user is structured and has user-supplied identifying names associated with each item and group. These names are returned via the interrupt executive (described later) when light pen or tablet hits on the display are recorded.

A report type call, Itemize, allows a user to interrogate an existing display file to discover its structure. This call is useful when working with a display buffer file built by another program.

2. Implementation

Each active user is displayed in a round-robin fashion with a delay between the last and first synchronized to sixty cycles to reduce display hum effects. A push down list of names is formed each frame to reflect the hierarchy of the structure. If a pointing device is active (pen or tablet), a display feedback list of hit information is formed during the frame. Every effort is made to keep the scope running while changes to the display structure are made.

The scope hardware in operation for the past three years has no character generator; all characters are interpreted and drawn as line segments by the display executive. Software edge detection for character overflow is also contained in the executive. New hardware provides both character generation and character edge overflow.

The display structure is created and maintained by the display executive which handles the free storage, garbage collection and ring structuring for each file. Each display file is initially three pages (768 words) long and is expanded when needed up to twenty pages. Since the display file is frozen in core, expansion is done only when necessary and then only to the amount needed. Rings of user-given block identifiers are also kept to minimize the search time during file manipulation. The blocks and linkage of blocks in the display file were designed specifically to accommodate the data channel hardware.

3. Conclusions

The presen display executive is sufficiently flexible to handle satisfactorily the needs of most users. The most frequently used display program is the character editor associated with the on-line assembler, the VITAL system, and other text manipulation systems. On-line scope character editing is a way of life now that few users will do without. Other display usage is for Reckoner system plots, debugging displays, and general graphics applications.

In retrospect, the design decision causing the greatest problems was to have a user pass display data to the executive as specific hardware bit commands. This early decision was made to facilitate the conversion of existing programs to display operation in time sharing. Operation was satisfactory until (a) second generation display hardware was installed, and (b) a remote display console on phone lines was attached to the TX-2. The use of a specific hardware oriented display data form then became most awkward. Special conversion routines were needed to translate display data oriented to specific hardware into suitable forms for both the network display and new hardware generators.

Other problems in the executive contribute to this difficulty. An appropriate solution would appear to be the use of a general graphic data format about positions, line and curve parameters, and characters. The display executive would then calculate the appropriate hardware or network data formats from this general form. An APEX restriction, however, is that all calls to the executive run to completion and are not broken on time slice completion. This makes extensive executive calculations impractical and determines the APEX mode of incremental information calls. A way around this restriction is available and used for the conversions necessary from existing formats to the new hardware and network requirements. This method, however, has high overhead and involves the reactivation of a system program in user mode by the executive which can then be broken for time slicing.

Experience with the APEX display executive has indicated the desirable characteristics of a time-shared display system. The basic existing facilities are satisfactory with the major exceptions noted above. Time-sliced calculation facilities are needed to handle a hardware independent data representation as well as to permit the checking of large data transfers to prevent input-output hang ups. While it is possible to accomplish this in the present executive, its provision in the initial design would result in smoother operation.

B. APEX Interrupt Executive

The interrupt executive is a portion of the APEX executive which handles on-line input information from all console input devices including those which don't provide hardware interrupts such as knobs and toggle switches. These are included to remove the burden of constantly sampling registers from the user. The devices which do not cause a hardware interrupt are sampled by the executive on the basis of an interval time interrupt. Information is given to the user only when a change of state of the requested device occurs. Light-pen tracking (which is difficult and consumes excessive display time) has been eliminated by the use of the tablet.

The interrupt buffer which contains the user's information is an extension of the keyboard buffer, a stay-in-core file, enabling the executive to store information for a user ever though his programs may be on the drum. The buffer is circular with each stack of information identified with the device which caused it and a mask stating which devices are being reported in the stack. This enables a user to obtain information quickly and allows packing of data since registers need not be pre-assigned to devices. The buffer can hold up to 256 registers of interrupt information. Even though extensive software interlocking was required, as many sequences as possible were used to minimize loss of interrupts.

These few techniques by no means solve the response problems, since the user's program must be brought in from the drum to process the information. The hope is that once he is brought in, he has enough information to process to make the swap time worthwhile. The question of transmitting interrupt information back and forth to the remote consoles has been considered but not yet completely resolved. The interrupt executive is still relatively new and needs further use for a realistic evaluation.

C. APEX Scheduling Algorithm

At the beginning of the year, a major revision of the APEX scheduling algorithm was undertaken. Prior to that time a two-queue round robin algorithm had been in use. The "fast" queue had been intended to give good response for interactive tasks, and the "slow" queue had been expected to handle the bulk of the computing load for tasks where response requirements were not critical. In practice, it had developed that many tasks considered by the users to be interactive posed computational loads which put them on the slow queue, and at the same time there were a significant number of large jobs in the mix which by nature should have been treated as background but were being given equal status on the slow queue. For many users, the behavior of the system was indistinguishable from that expected from a simple round-robin scheduler with a slow switching rate. Performance was inadequate, both with respect to response time and computational efficiency. The new scheduling algorithm described here achieves improved performance by making a finer discrimination among the tasks. Instead of two queues separated by an arbitrary threshold, there may be as many queues as tasks. The whole range of tasks, from fast interaction to background, can be handled, with each task competing for processor time against tasks of a similar nature.

In the current APEX scheduling algorithm, a task is always in one of three states: running, active, or inactive. A running task is the one actually executing instructions. An active task is one waiting for processor time and/or core space. An inactive task is waiting for some

external signal, either an input-output device or the action of a human user. The scheduling algorithm is concerned with deciding which active task should be run next and for how long. In APEX this decision is based on the demand for processor capacity which the tasks have indicated by their behavior during the last few minutes preceding a scheduling decision. The scheduling algorithm is designed to give better service to least-demanding tasks. When a running task reaches the end of its time slice, the least-demanding task on the list of active tasks is chosen to be run next. The new task will be given a time slice computed from its demand value.

Ideally, a measure of demand would take into account CPU time used, core requirements, and input-output load. Demand, as measured in APEX, is based primarily on CPU time used, with some adjustment for core requirements, but with no direct inclusion of input-output load. The units of measure are instruction executions rather than real time, since the rate of instruction execution depends on the input-output load which, in turn, is affected by the other tasks in the system. A timer in TX-2 counts the instructions executed in the task program as well as in supervisor programs called by the task. In discussing the measurement of demand we will use equivalent time values rather than executed instructions, because the units are more generally familiar. The time values correspond to instructions executed in the absence of input-output load. Real running times are generally longer, significantly so when there is heavy display activity.

The single number representing demand increases when a task runs and decreases at about the same rate when it is inactive. Thus an interactive task characterized by infrequent short bursts of activity will have a low average demand value. When such a task moves from the inactive to the active state, a change is made to its demand value which is proportional to the number of memory pages which must be transferred from secondary memory to core memory in order to run the task. Interaction with a large program will therefore be interpreted as a more demanding task and service will be slower. A moderate computational task such as a compilation requiring 20 seconds of CPU time will build up a demand value which will effectively suppress it in favor of more interactive tasks. Often, when the compilation is complete, the user will want to proceed with some interactive activity, but the demand value accumulated as a result of his compilation will tend to cause him to be scheduled unfavorably. An arbitrary reduction of the demand to a very low value when a running task becomes interactive would assure good performance in this situation, but if another long computation were immediately initiated, the momentary inactivity would give the task an unfair scheduling advantage. As a compromise, demand values in excess of about 5 seconds are reduced to that value when the task becomes inactive.

For a really long computational task, the demand value could become very large, so that it would become highly probable that a mixture of small and moderate tasks could keep the long task out of the processor indefinitely. To prevent this situation from developing, the demand value is given a maximum upper limit corresponding to about 40 seconds of CPU time, and in addition the demand value for active tasks waiting to be scheduled is reduced slowly with time. As a result, every active task is guaranteed a fair share of the processor over a period of several minutes.

When the APEX scheduler chooses to run a new task, it selects the active task with the least-accumulated demand value. It gives that task a time slice equal to the accumulated demand

and begins running it. At this instant the known core requirements of the task have been met and it is runnable. From the scheduling point of view, one of three kinds of events will occur. First, the task may run until the end of its time slice, in which case the basic scheduling process will be repeated. If the task in question is still the least demanding, it will be run again, this time with a time slice twice as long. Doubling of the time slice continues until a maximum value of about 13.5 seconds is reached. This value corresponds to the maximum count possible in the timer. From the point of view of response time, it would be desirable to limit the time slice to a much smaller value. Unfortunately, the ratio of typical user core requirements to available core supply makes it probable that there will be idle time in switching between large computational tasks. To keep CPU efficiency at a reasonable level, the time slice must be longer than the idle time. With the present core size and secondary memory speeds, as much as seven seconds may be required to switch between two large users. The 13.5-second maximum time slice appears to be a reasonable compromise.

The second kind of event from a scheduling point of view occurs when a running task becomes unrunnable as a result of its own actions. If this situation results from a request for input-output which has not yet been satisfied, the task is placed in the inactive state and scheduling of the remaining tasks proceeds as if the task had used its time slice. If the task has become unrunnable because of a need for a secondary memory transfer, the scheduler computes an estimated cost for the transfer in equivalent instruction executions, adds that cost to the amount used in the current time slice, and if a reasonable time remains, initiates the transfer. During the hundreds of milliseconds required to effect the transfer, the scheduler will attempt to find another runnable task, which, if found, will be run in a special freeloading mode. At the completion of the transfer, running the originally scheduled task will be resumed for the remainder of the time slice. The scheduler always attempts to find a potential freeloader, using the same principle of choosing the least-demanding task, if the primary task being scheduled is not immediately runnable. Obviously, the scheduler must not allow a freeloading task to make use of the secondary memory data channel or its commitment to the primary task will be subverted. The policy of honoring requests for secondary memory transfer from primary tasks is essential if the computational efficiency advantages of long time slices are to be achieved. It is important in this event to make an appropriate charge for the transfer to prevent a task which makes large use of secondary memory transfers from remaining the primary task for too long a time.

The third important event from a scheduling point of view occurs when a task becomes active whose accumulated demand value is less than that of the running task. In this case the scheduler allows the new task to pre-empt the processor, but not until the running task has been allowed to use an amount of CPU time equivalent to the time slice to be given to the new task. To avoid unnecessary overhead, this new time slice is never allowed to be less than a minimum value, currently about 1/6 second.

The new APEX scheduling algorithm is reasonably satisfactory in balancing response time against computational efficiency. Obviously, absolute response time is a linear function of secondary memory access and transfer time. The TX-2 secondary memory operates at disk speeds and response times are often not as good as the user would like. The only alternative presently available is to keep the responding programs in core memory, and this technique is

used in a few special situations, but it is obviously limited in application because core memory is limited. Computational efficiency is currently limited by the large ratio of core required by task programs to available core. If this ratio could be kept below one half, there would be a reasonable probability that more than one runnable task would be in core at one time, and the computational efficiency could be brought up by reducing idle time. This ratio can be improved by reducing the core requirements of task programs or by adding core to the system. The latter course is apt to be more economical, since the task programs are predominantly large software subsystems like compilers and assemblers which are not easily reworked to use less core space. It is often much less expensive to buy more core than to pay for the large programming effort required to improve the software.

From the point of view of the user of a large interactive subsystem such as many graphic application packages tend to become, the task scheduling here is completely inadequate. The difficulty arises because the task involves both major computational loads and fast user interaction. To handle this sort of task at all satisfactorily, it must be split into two subtasks from the point of view of the scheduler. The interactive subtask can then be scheduled in a fashion appropriate to its requirements for response time,[12] and the computational subtask can be scheduled for computational efficiency. If the task can be split, the current scheduling algorithm should give acceptable performance. Changes in other parts of APEX will be needed to accomplish the task splitting in such a way that adequate communications between the subtasks can be obtained.

D. PDP-338 Display Console

A 338 remote display console has been attached to the TX-2 computer in order to explore problems of networking computers and the exchange of graphic display information. The intent was to have the 338 serve as just another TX-2 console. Display programs which run on local display consoles should run without change on the remote console. The 338 is now operating routinely in Washington, D. C., with the capability of a local TX-2 console, and most current programs can be run remotely. Thus the goals have been partially met, and the experience of creating the remote console has uncovered many problem areas.

The greatest difficulty in implementation was caused by the nature of local TX-2 input-output. The TX-2 does its various console outputs in an overlapped and highly parallel fashion, thus making it difficult to modify the executive in order to funnel all these outputs to a single sequential remote line. In effect, each output section of the executive had to be modified to check specifically whether the user requesting output service was remote, and if so special treatment was provided. If remote graphics is to be at all practical, executive systems should be appropriately organized. Service requests should be sequential and passed either to the remote station directly or through a pseudo link to a local section of the executive which mimics the remote.

The return of information to TX-2 from the remote console suffers from the same sequential-parallel problems. Many of these input problems are related to the input executive and have yet to be resolved.

The final problem area of interest deals with the data formats exchanged over the link. As an expedient, the graphic data sent to the 338 reflects the particular structuring allowed in TX-2

APEX displays; this format is by no means general purpose. A related problem of critical importance to future networking efforts involves methods for describing and documenting whatever formats are used. Either a standard data format must be devised and its use enforced (unlikely), or ways must be found for the clear description of a variety of formats.

II. LANGUAGES

A. VITAL Compiler-Compiler System

1. Introduction

VITAL I was implemented on TX-2 to provide a means of constructing compilers for a variety of problem-oriented languages. Since the languages were to be experimental, the primary design objective of VITAL was that compilers be easy to modify in their on-line environment. Little or no attention was paid to other compiler-compiler design considerations such as the formalization of language definitions, machine-independent compiler description, incremental or multipass compilation, compilation speed, object code efficiency, or noncompiler applications.

A VITAL compiler consists of a parsing algorithm written in a recognizer language FPL (Formal Production Language) and a number of associated semantic routines written in a semantics-oriented language FSL (Formal Semantic Language). The FPL and FSL specifications for a given compiler are translated by two special purpose compilers. A source program can then be compiled by executing the resulting parser, which in turn executes the necessary semantic routines during the scanning of the source text.

2. Compiler Syntax Specifications

An FPL program consists of two sections. The first specifies how the raw source program character string is to be broken into basic syntactic entities (words) and lists the words which have special syntactic significance (reserved words). These rules govern the preprocessing which the VITAL system does on source program text when the text is entered. The second section contains the production statements. Each statement specifies a pattern of syntactic entities and a list of actions to be taken when that configuration is matched.

An FPL program acts as a recognizer with a single push-down stack. When a word (as determined by the specified preprocessing rules) is requested from the source text by a scan command, the next word is pushed onto the stack. If the configuration at the top of the stack matches the pattern designated in the production statement that is being executed, the actions specified by that production are performed. These actions may include transformation of the stack configuration, execution of a semantic routine, further scanning of the source program, and transfer of control to another production. If a production fails to match, the next sequential production is executed.

3. Compiler Semantic Specification

An FSL program consists of declarations of compiler data structures and a set of semantic routines. Each routine is a description of the semantic actions to be taken when some language

construct is recognized in the source program. These actions include: recording the changes in the state of the run-time processor which are implied by a source program construct, generating code, performing internal bookkeeping tasks, and generating compiler output such as formatted listings or error messages. The necessary table and stack manipulations and the generation of code and printed output are done in FSL with special data structures and primitive operations.

The most important feature of FSL is the code generation facility. Enclosing an expression or statement in 'code brackets' indicates that the expression or statement is to be compiled into object code rather than executed during the compilation; the operands of the statement or expression are interpreted as semantic descriptors of run-time operands (designating the address, data type, etc., of the operands of the run-time code). The compile-time result of a code-bracketed expression or statement is the semantic descriptor of the run-time operand (if any) which will result from the run-time execution of the object code compiled. This result descriptor may be discarded or recorded for later use as an operand of another code-bracketed expression or statement.

4. Conclusions

The VITAL I system has met its major design goal by providing a flexible on-line system for language experimentation. VITAL is unique not as a compiler-compiler but in the way it matches its on-line environment. VITAL interfaces with the scope and keyboard editors in a natural fashion and also can be used to call any of the APEX public routines. Switching between compiling a user program and editing the productions or semantics of its compiler is simple. The semantic description of a compiler can contain sections for formatted listings of source programs and as fancy error messages as the compiler designer chooses to include. These extra service features are a necessity, since compilers which make inscrutable code and return cryptic error messages are not very useful. VITAL I provides the compiler maker tools with which he can both compile code and interface smoothly to the users of his compiler.

The weaknesses of VITAL are peculiar to its implementation and occur largely from its irregular growth. These problems are now locked into the present system because the major efforts required to fix them would have to be diverted from the development of a second generation system. The code generation features of FSL, while useful, require major design revision; this task is now under way for VITAL II. Code generation for run-time input-output is also deficient. The production syntax scheme is adequate, but more flexible control of the scanning of source text is desirable. VITAL itself is large and occupies more core than would have been necessary if it had been designed with partitions in mind. A more detailed exposition of these problems and their recommended solution is contained in a report now in preparation.

VITAL has been used to create several demonstration compilers. In this respect it would make a good teaching tool. Several compilers in regular use have also been made including a PDP-8 assembler, a circuit-testing language compiler and, most notably, the LEAP compiler discussed elsewhere. VITAL I is also being used to design and bootstrap its successor VITAL II.

B. Language for Expressing Associative Procedures

There is much interest these days in using a computer to:

Store a large number of relations about some collection of items,
Answer questions concerning these items and relations,
Dynamically add (delete) items and relations.

For example, consider the design of a simple system for the computer-aided manipulation of two-dimensional line drawings. Input information would be taken from a light pen and push buttons, and output would be shown on a computer-driven display. We would allow only points and lines as parts of a drawing, and we would represent drawings inside the computer by remembering all points and lines and all relations of the forms

"STARPOINT OF LINE IS POINT,"
"ENDPOINT OF LINE IS POINT," and
"COORDINATES OF POINT ARE (X, Y)."

To display a picture, we would ask the computer for the coordinates of the startpoint and endpoint for each line, and then pass this information to display output routine. To erase a point we would delete the point from the store of items, and delete all relations involving the point.

For a useful computer-aided solution to problems which deal with such a dynamic associative information base, there are two basic requirements:

(1) A convenient way for a human to express his instructions on how to solve the problem,
(2) An efficient way to represent the information base inside the computer.

Ease of expression of such problems requires the facility of a high-level computer language. Within the framework of this language, a user's conception of the representation of his problem should be as natural as possible. In general, this requires that he be insulated from the details of implementation wherever practical. In addition to a facility for expressing manipulations of the dynamic information base, this language must have algebraic capabilities, and the other standard algorithmic facilities of an ALGOL, or a FORTRAN. Also, interactive capabilities must be available.

The LEAP language is based on an extended algebraic language similar in form to ALGOL. The extensions include sublanguages for display-output and homogeneous matrix-manipulation, and a facility for interactive input-output. The compiler for LEAP programs was implemented using the VITAL compiler-compiler system on TX-2. LEAP programs are compiled on-line, in an interactive, time-shared environment.

The constructs in LEAP of particular interest deal with its data-structuring techniques. The basic data structure entity used is an associative triple of the form "Attribute of Object is Value," (e.g., Component of Steamsystem is Pipejoints). The data structure is a store of facts in this form. Programming constructs are available for creating, deleting, and searching for elements in the data store. Of particular importance is the uniformity of the single data form used. A LEAP programmer does not have to consider the details of a complex structure in computer memory; he can concentrate on what he wishes to represent and not how to represent it. The

programming facilities available include set variables and operations, a powerful search and retrieval facility, and the ability to use a triple itself as a component of another triple. For example, the following set of facts is allowable:

XX: CAPT of FORRESTAL is SMITH
APPOINTMENT of SECNAV is XX

In many commonly used data structuring techniques, it is hard to link a pointer to the fact that two other pointers are related.

The LEAP language has been extensively used for a variety of programming tasks. The integrated circuit work described elsewhere in this report is entirely written in LEAP as are the experimental constraint satisfaction programs. Other LEAP programs include several demonstrations and experimental programs used to verify three-dimensional curve routines. Two graphical debugging programs are written in LEAP as well as portions of the new VITAL system. The flexibility of the LEAP compiler due to VITAL has been most advantageous. LEAP has grown and expanded in response to user requests and experience. New constructs can be added at weekly intervals with a minimum of effort. Furthermore, LEAP has helped to refine VITAL as the needs of this large compiler are reflected back to the compiler-making system.

LEAP at the moment is weak in its ability to link to other programs and to use and form parts of a library. It also suffers from the problems of size which it inherits from VITAL. Debugging facilities are at present scanty although it is worth remarking that very few programming mistakes are made. Such mistakes are generally obvious once the nature of the error is determined.

Part of the LEAP data structure design allows the segmentation of data structures which are too large for active core memory. Strategies for efficient accessing of such a partitioned data base are planned but have not been tested.

C. Debugging

1. On-Line Trapping and Debugging

Mark 5 is the assembler operating system in the APEX time-sharing system. From its early days, it has served as a test vehicle for the development of techniques for debugging machine language programs.

Most of the debugging techniques are made possible by the trapping hardware of the TX-2 computer. In effect, this hardware acts like an input-output channel, or sequence, which can be made sensitive to certain conditions in the user-program sequence. One such condition is the breaking of normal program flow caused by jumps and skips; others involve the referencing of marked instructions or data registers.

The trapping hardware provides a unique method for placing breakpoints on registers of a program. Each memory register in the TX-2 computer has an extra bit, the metabit, which, under timesharing, is used exclusively for breakpointing. The trapping sequence can be made sensitive to metabit tagged registers when they are encountered in any of the three basic machine cycles: instruction fetch, data fetch, indirect address fetch. The first provides the familiar breakpoint facility without any modification of the running code. Data fetch breakpointing permits

trapping on references and modifications to data cells. Indirect address trapping is useful for debugging pure procedures which obtain their data via indirect addressing. Data and indirect address breakpointing are generally not available in all-software debugging systems.

After a trap occurs, information relevant to its cause is provided by the hardware. The role of the debugging system is to decode this information and provide a meaningful symbolic message to the programmer. By utilizing the paging facilities of the TX-2 computer the time-sharing system provides different virtual memories to the debugger and to the program being debugged. Thus, with a small amount of software manipulation, the program interrupts caused by trapping are completely transparent to the program itself. It is also important to note that between traps, the program runs at full speed.

In another mode jumps and skips are monitored by the trapping hardware and the from-to address information is collected by the debugging software in a circular buffer. Should difficulties later develop, recent program history is available for backtracking purposes in debugging. The cost of operating programs in this mode is an average slowdown of about four or five which is far better than could be achieved through interpretation.

The software operating in the trapping sequence need only record the information supplied by the hardware, rather than generate it through interpretation. Since the sequences are autonomous, this can be done efficiently and without affecting the user program.

Other uses of trapping on "modified program counter" have been trace-like dumps and flowmap displays. The latter is a rather novel approach to the presentation of program flow description. An on-line display presents a global view of the program flow rather than a sequential trace dump. A typical flowmap has nodes representing locations in a program, curved lines between nodes for jumps or skips, and straight lines for sequentially performed sections of code. Facilities are available for compressing areas of code into single nodes, thus providing a flowmap of routines rather than single instructions. Using flowmapping, one may quickly determine which parts of a program have been exercised and how control has been passed on an inter-instruction or inter-routine basis.

Another hardware sequence, the main alarm, manages the time-sharing privileged instruction violations and memory protection errors. When an error occurs, the main alarm traps and provides the relevant information to the debugging software which decodes this information and produces a symbolic message to the programmer.

In summary, the following are the hardware and software debugging facilities presently available on the TX-2 computer through Mark 5 and the APEX time-sharing system.

Hardware

Metabit trapping modes: instruction fetch, data fetch, indirect address fetch

Modified program counter trapping

Trap on all instructions (not currently used)

Complete reporting of cause of trap

Complete reporting of time-sharing privileged instruction violations and memory protection errors.

Software

Breakpoint trapping

Trace-like dumps

Flowmap displays

Monitored program execution ("spy") and associated backtracking information

Detailed reporting of cause of trap or error

Symbolic output in above facilities

Examination of program and machine state registers in any of several formats, including symbolic instruction format.

The above facilities have thus far been oriented toward the debugging of programs written in assembly language.

The debugging software has, as an experimental expedient, been added to the Mark 5 assembly system to make easy the conversion of trap data to symbolic program references. A more general stand-alone debugging package is under development. This new debugging program will be driven by symbolic information obtained from any compatible compiler or assembler. Breakpoint trapping carries over well from assembly code to higher level languages. Labels are clear places at which to trap. The carryover on data trapping is more complicated. The data type of variables must be known to the debugging program so that values can be provided appropriately (i.e., Boolean as T or F vs 0000000001). Complex data structures (i.e., arrays) require a still more detailed specification of the data trap conditions than the single register variable normal in assembly code.

Further extensions to the present trapping capability are clear. When a user declares the conditions for a trap he should also declare what he wants done when the trap occurs. For example, the simplest way to accomplish this is to associate a program with each trap declaration. Generally these programs will be simple and can be created by statements in a debugging language. However, any program should in principle be invokable by a trap. In particular, the user should be able to specify what information he wants printed out at the time a trap condition is declared. He should also be able to specify test conditions to provide conditional trapping and also trap for statistic gathering purposes with manual intervention unnecessary on each trap.

The continuing development of these debugging facilities has involved a mating of hardware and software into a unified system where the role of each is dictated by its ultimate capabilities. Unfortunately, all too often, debugging systems are left as a software exercise in an alien environment. The efforts of this research are designed to ease this problem by producing information on which an integrated translating-debugging system might be based both at machine language and compiler language levels.

2. Debugging and Complex Data Structures

a. Introduction

This section reports on some novel approaches to problems in on-line graphical debugging that arise when programs manipulate complex data structures. The term "on-line" implies that

the programmer is in an interactive environment with both his program and a debugging system. He may suspend the execution of his program and examine it with the aid of the debugging package. When execution is resumed, all trace of the debugging activity can be made to disappear. The term "graphical" means that the debugging program may utilize visual displays to convey information about the program's state and that the user may utilize a light pen or analogous device in specifying commands to the debugging system.

In a 1965 review article, Stockham notes that past debugging experiments focus on "the display of the dynamics of program state" and "the interrogation of program variables in a variety of formats."[1] Evans and Darley in a 1966 survey paper stress that the examination of program and data should be "in terms of the notation of the language of the program."[2]

A major weakness in current debugging systems is that their descriptions of program state are lagging five to ten years behind the corresponding descriptions in programming languages. State interrogation commands in today's debugging packages seem limited to referencing single memory locations or blocks of contiguous locations. Other commands facilitate reaching "frequently desired states... from the present one (e.g., look at the contents of the current register ±1, look at the contents of the register addressed in the current register)."[2]

These memory referencing capabilities were adequate when programs were manipulating only simple organizations of intermediate results, for example, scalars and unidimensional arrays. Even in the latter case, since the elements of a vector are usually stored in contiguous memory locations, existing tools allow facile interrogation of a vector's state during the suspension of a program.

Multidimensional arrays and matrices are usually also stored in contiguous memory locations. Some simple mapping function f transforms MATRIX (I, J, K) into MATRIX1 [f(1, J, K)], where MATRIX1 is a linear arrangement of the three-dimensional array MATRIX. Knowledge of this function enables a programmer to interpret without overwhelming difficulty a linear dump of the array's storage.

If, however, the number and size of the arrays is to be determined or altered during program execution, run-time storage allocation becomes an issue of consequence. A garbage-collection routine must keep track of free storage and reassign that which becomes available for reuse. Expansion of an array beyond the bounds of the area reserved for it may necessitate a lengthy copying procedure. The concept of pure contiguous storage is therefore usually abandoned and replaced by that of storage in blocks of contiguous memory, each of which is linked by a pointer to the next block. The programmer examining a memory dump of such a structure needs a description of the organization of data blocks and pointers to trace through it.

The use of list structures in representing complex multidimensional associations among entities significantly aids applications such as graphics and natural language processing.[3,4] Yet the use of list structures requires cumbersome planning of a detailed organization of data blocks and pointers. Efficiency may degenerate owing to the run-time overhead of chasing pointer strings. Hence researchers have sought new techniques with which associations among items of information may be represented.

One such technique is hash-coding, in which items of information and the associations among them are stored in memory locations selected by arithmetic functions on the internal

identifiers of the elements. We assert the claim that in this context classical debugging tools are inadequate for discovering quickly which associations are present in the data base of a program.

In summary, the existence and increasing use of complex data structures has generated a corresponding need for new debugging tools. A basic debugging operation, the interrogation or examining of a data base, can no longer be adequately performed by referencing single registers or single contiguous blocks of core. We shall report on two pieces of recent work which have resulted in operative debugging programs on the M.I.T. Lincoln Laboratory TX-2 computer. The first program is used to examine list-structured display files; the second is used to interrogate hash-coded associative data bases.

b. Interrogation of List-Structured Display Files

TX-2 display files are organized into a list structure which is a subset of the CORAL data structure. As the historical evolution of this approach to the storage of graphical information has been documented elsewhere, it suffices to summarize here the nature and interpretation of allowable structures.[5]

Sets of display elements, such as points, lines, and text, are stored in blocks of contiguous core. These blocks are inserted upon user command into rings in which each block is linked by pointers to successor, predecessor, and ring start blocks. Each ring of elements may be assigned a unique origin on the scope face and a unique sensitivity to a pointing device such as the light pen. Individual rings may be linked through a special kind of block known as the use block. The presence in ring 0 of a block which is a use of ring 1 indicates a hierarchal relationship in which the latter is in some sense a substructure of the former. Finally, the contents of only those rings which are directly or indirectly linked to ring 0 are displayed. There may be several use blocks of the same ring in a structure.

Let us now imagine that a TX-2 user is constructing a display for the first time and that some portion is missing. What can be wrong? Has the calculation of the missing part gone astray or not been executed? Has it been successfully executed, but its ring not linked to ring 0? Visually, it is impossible to judge.

Baffled, the user takes control from his program, which presumably had ceased calculating and was only showing him the resulting picture. He then calls the interrogation package DISPLAY DISPLAY. It accepts his display file as data, analyzes the ring and use-block structure, and produces a display of that structure. The user may then request a further display of the (essentially unordered) set of blocks within a particular ring, either by typing its identification number or by pointing to a corresponding node of the tree. Finally, he may ask to see the contents of a particular block. He can thereby distinguish any of the possibilities in the above instance.

c. Interrogation of Hash-Coded Associative Structures

Hash-coding is a technique with which one can represent complex collections of entities and relationships among them. An implementation of a hash-coded data base and associative retrieval language is the LEAP Language for the Expression of Associative Procedures.[6] To represent a

relationship between three items of information, one combines arithmetically by some standard mapping function the internal identifiers of two of the items and thus obtains a unique memory location. The identifier of the third item is stored in that location. Problems generated by the many-one nature of hashing functions, the partition of large data bases in auxiliary storage, and the criterion of efficiency require that the implementation make use of list structures as well. Once again we stress that classical debugging techniques are now inadequate, for a programmer must perform a hash-coding and trace a list structure to discover whether or not a particular association is present in his program's data base.

A LEAP associative structure consists of a universe of items of information and a universe of associations which are triples of these items. Items may be referenced directly by name or indirectly in terms of the relationships (triples) in which they appear. Hence LEAP facilitates associative retrieval requests such as "find all X, Y, and Z in the data base such that the 'TYPE' of Z is a 'LINE' and X is an 'ENDPOINT' of Z and Y is an 'ENDPOINT' of Z." A typical LEAP statement in which this expression could appear is

```
FOREACH TYPE.Z = LINE and ENDPOINT.Z = X and ENDPOINT.Z = Y DO
    BEGIN
    MAKE COLLINEAR.X = Y;
    MAKE COLLINEAR.Y = X;
    END
```

The effect of this statement would be to augment a data base, one in which the points, lines, and line-endpoint pairs are identified, by those relationships of collinearity that may be deduced from the association of two points forming the endpoints of a single line.

There are seven primitive associative forms corresponding to the specification of one, two, or three places in a single triple. Their fundamental character is mirrored in the debugging package; corresponding to each form is an interrogation function. Again we imagine that the user interrupts his program and requests the use of the debugging program. The LEAP INTERROGATION PACKAGE presents him with a global display of the program's data base. The process of interrogation consists of defining a succession of calls on interrogation functions to yield debugging displays of aspects of the data structure.

For example, the function of two arguments "_._ = X", when applied to the item "ENDPOINT" and the item "LINE 1" (the name of a line in the data base), yields the collection of all endpoints of LINE 1. These are those items appearing in the third position of triples in which ENDPOINT occupies the first position and LINE 1 occupies the second position. Similarly, the function "_.X = Y", when applied to the item "COLLINEAR", yields the ordered pairs of all items appearing in triples in which COLLINEAR occupies the first position.

A succession of calls on interrogation functions allows the user selectively to examine portions of the associative data base. Each new debugging display is saved on a stack of such displays and is available for later reviewing. Items appearing as results of function calls are usually used as arguments to further interrogations, and in this manner a programmer can sequence through the interesting regions of an arbitrarily complex structure.

In LEAP, the ability to couple classical data types such as real scalars or integer arrays with the relational structures is achieved by allowing an item to possess a value of arbitrary

(ALGOL) data type. Interrogation requests formulated in a manner identical to the mode of reference in LEAP will yield in response the values of certain items.

III. GRAPHICS AND APPLICATIONS

A. Earth Display Program

1. General Description

To aid in the development of optical detectors for use in various control systems, the LES-4 and LES-5 satellites each carry equipment for earth-albedo experiments. Evaluation of the results in two specific areas of these experiments was aided by the development of a computer graphics program: first, determination of the spin axis of LES IV and second, location of the earth-albedo points.

The earth display program displays a projective transformation of the earth as seen if the observer were sitting on the satellite. Information also automatically displayed includes the actual time at which the satellite was in this position, the incremental time to be added for the next displayed frame, and the right ascension and declination of the spin axis in celestial space. Under control of the typewriter is the activation of a variety of options available to facilitate use of the program. The actual track of an optical sensor swept out in one complete satellite revolution and the exact position of the sensor are available on command. Since that part of the earth which is illuminated is of interest, a sun-shade line together with a small "sun" indicating which part of the earth is illuminated is available. Upon command the earth will automatically rotate to match the movement of the satellite in the orbit. The rotation may be moved forward or backwards in time at any rate desired. The time of each frame, accurate to 5 ms, is also displayed.

The mathematical techniques used to construct the display were derived from the paper, "Homogeneous Matrix Representation and Manipulation of N-Dimensional Constructs" by Lawrence G. Roberts of Lincoln Laboratory. To minimize the time of computation, the entire earth is represented by about 120 points in 3-space. In constructing the display, all the projective transformations are combined into one matrix operation requiring only one matrix multiplication per point of the earth. The result of the transformations places the image plane at $x = 0$, allowing the points lying behind this plane to be eliminated easily.

Given the orbital parameters of the satellite at some time t_o, the position of the satellite in the elliptical plane of the orbit at any time t is determined using the methods discussed by W. M. Smart in Text-Book on Spherical Astronomy (Cambridge University Press, Cambridge, England, 1944). To compensate for the atmospheric drag on the satellite when perigee is approached, first and second derivative corrections are applied to computation of the angular velocity of the satellite measured over one period of satellite rotation. The right ascension of the ascending node of the orbit and the argument of perigee measured in the orbital plane also have first-order correction terms. No other Taylor approximations are incorporated in the computation of orbital parameters.

2. Results

The most important accomplishment was actual use of the program for the purpose for which it was designed, i.e., determination of the LES-4 spin axis and location of earth-albedo

data. Currently, the program is being used on LES-4 data for validation and plans are being made for runs with LES-5 data.

One of the primary obstacles in the development of this program was insufficient communication with the engineers who were to use it. In particular, such information as the best units in which to express orbital parameters and other relevant satellite data was not known early in the program's development. As a result, much change of input and representation was required after the program was working. The only constraint that the TX-2 time-sharing system places on the program is degraded response when other users are on.

B. Constraint Systems

1. Introduction

The aim of this work is to model the behavior of systems of interrelated variables, especially those arising in computer graphics and engineering design. The components of such a system are a set of variables and a collection of relations among these variables. The relations may be arbitrary; in particular, each variable is usually not specified as an explicit function of others.

Such implicit systems, subsequently termed constraint systems, are more general than systems in which unknown output values are explicit functions of known inputs. The primary questions of practical concern are:

(a) Suppose values of a subcollection of the variables are given, that these values are mutually consistent, and that they are sufficient to uniquely determine the remaining variables, though they do not explicitly do so. How can these variables be efficiently determined? Are there methods more effective than relaxation-type techniques?

(b) Suppose that the given values are not mutually consistent, i.e., the system is "overconstrained." Can the values be singled out which are in some sense "the root of the trouble?"

Answers to these questions have been provided by the abstract model and algorithms discussed below. These algorithms are applicable to any system representable by the model. They have already been implemented by programs accepting fairly wide classes of systems as input.

2. Mathematical Model

An n-dimensional geometric structure has been developed capable of modeling a variety of constraint systems.[7] The number of degrees of freedom in the system has been shown to be related to the dimensionality and connectivity of the model.

For the sake of practical application, the primitive relations that have been considered are addition, multiplication, vector addition, scalar multiplication, and rotation. These relations have been modeled by elementary k-dimensional "cells". A discussion of these k-cells is deferred to a more complete exposition. The constraint system itself corresponds to a structure built of these cells. Salient features of the system have been found to have simple structural analogues. For example, analytic symmetries in the system equations correspond to geometric symmetries in the model; these symmetries permit structural reductions that facilitate the determination of a solution when one exists. Logical consequents of the system equations are

related to substructures identifiable by straightforward operations upon the model; these substructures "factor" the model into easily solvable parts.

3. Programmed Applications

The above model has been implemented in two interactive graphics programs. The first simulates a common mechanical linkage, the pantograph. The second program, more powerful, permits simulation of arbitrary resistive networks. Though apparently dissimilar, these two types of systems are simulated by the same constraint model.

Solutions are obtained by interpreting the constraint structure, rather than by methods of numerical iteration; the result is in effect an analytical solution. Hence it should be possible to compile solution-yielding subroutines rather than particular numerical answers. Such an approach would produce fast repeated solutions, even for highly constrained systems, once the process of determining and compiling the analytical solution is completed.

4. Overconstrained Systems

A method of treating overconstrained systems has been developed. Minimal subsets of overconstrained variables have been characterized, as well as an algorithm for discovering them. Elimination of an overconstraint for a minimal subset has been shown to be simple. Elimination of the system's overconstraint is reducible to a sequence of these simple steps.

5. Arbitrary Linear Systems

The techniques that have been developed and programmed are sufficiently general to handle arbitrary linear systems. In particular, they are capable of producing analytical solutions when such exist, even if the number of equations exceeds the number of unknowns. Such is the case with resistive networks, when one is given the whole collection of nonindependent Kirchhoff current-law relations and Ohm's law equations. By contrast, methods such as Gaussian elimination require data to be in a rigid n-equation, n-unknown form. Relaxation methods may converge slowly and cannot provide analytical solutions. The freedom of input tolerated by constraint techniques is particularly advantageous in interactive programming where data does not arrive in predigested form.

6. Localized Evolution

Change and transformation are general features of the engineering design process; therefore, a useful mathematical model should be well-suited to evolution. In particular, a local modification to a system should be reflected by a local modification in its model.

Such is the case for constraint structures. For example, given a linear system, a change such as "replace x_5 by $x_1 - 3x_5$" is accomplished by a single, local change to the structure. On the other hand, global alterations would be required if the linear system were represented in terms of matrices.

7. Parallel Processing

The constraint structure model is a dynamic one. It includes such ideas as propagation of values through the structure and equilibrium of cells. The "operation" of the structure is

essentially asynchronous and suggests possibilities of parallel processing. Work has yet to be done to determine whether this model could be useful in the design of hardware or algorithms for parallel processing The practicality of the model is demonstrated by the programs mentioned previously. They amount to software simulation of a procedure that would be naturally realizable by parallel processing hardware.

8. Generality of Model

The constraint model that has been described grew out of considerations of arithmetic relations. However, it has become clear that the mathematical features of the model are independent of arithmetic. The primitive idea is that of an effectively calculable relation. This observation leads to study of the model from an abstract viewpoint, as well as to applications apart from arithmetic. In particular, the model has been shown suitable for treating Boolean relations. Constraint structures can be built which correspond to complex collections of propositions. Possible logical consequents of such collections can easily be tested for validity. However, for want of both time and a good Boolean problem, Boolean applications have not been extensively considered.

9. Summary

Thus far this work has made possible: (1) the formulation of a constraint model capable of yielding analytical solutions to highly constrained systems, and (2) the creation of a strategy applicable to overconstrained systems.

Many questions have been raised; their range is suggested by the preceding paragraphs. In addition, issues of optimality need to be considered, and relationships with other methods need to be explored. Moreover, the problem of 3-dimensional constraints has yet to be considered; work in this area is being started.

C. Testing Integrated Circuits

In July 1967, it was decided that a superior approach to testing integrated circuits was to use the TX-2 rather than build a special tester for each type of integrated circuit. Moreover, special hardware testers leave unanswered many questions about failure modes and suboptimal test sequences. It was felt that with the flexibility of a general purpose computer, test patterns could be of arbitrary complexity and the test designer could choose test patterns on-line in much the same way circuits are probed with an oscilloscope

Therefore, a language called TIC (Testing Integrated Circuits) was specified. Its grammar is particularly suited to generating test patterns. A small hardware buffer and patch panel was built to interface an already existing input-output channel on the TX-2 to connectors for attaching the circuit under test. To aid the development of TIC programs, a debugger, called SOS, was written.

The job of writing a compiler for TIC was greatly simplified by the use of VITAL. By November 1967, the compiler, debugger, and hardware interface were operating. A test program has successfully been completed for testing and locating the exact faulty transistor or "via" in a 3-bit parity circuit. Test programs for other LSI circuits and for practical

nonexhaustive ways to test mo[illegible]mplex circuits are being written. For example, a defective 27-bit parity circuit might be useful as an n < 27-bit parity circuit.

TIC is a statement-oriented algebraic language. Expressions allow operations on subsets of Boolean test variables. For example, subsets of up to 36 variable sets may be permuted (by a shifting operation), complemented, or treated as a counter value and incremented. The handling of conditional expressions, functions, and program flow is similar to the Snobol III language. The normal test program outputs a set of "excitation" variables to the circuit, reads the "response," and compares the actual response with that expected.

The debugger, SOS, allows breakpointing and examination of data or program. Symbolic program references are statement-oriented (a compiled statement requires several machine registers). Plans are under way to modify LEAP to provide the symbol table necessary for SOS which may then be used to debug LEAP programs. SOS may also be used to debug machine-language programs.

D. Waveform Processing

1. Introduction

Oppenheim[8] has shown that the methods of filtering that have been applied so frequently to communications problems involving linear systems may just as easily be applied to a large class of nonlinear situations. This has been done by generalizing the principle of superposition through considerations of modern algebra.

The classical principle of superposition demands that if a system is excited by α and responds with α' and is excited by β and responds with β', the response to $\alpha + \beta$ must be $\alpha' + \beta'$. In generalizing this principle, the view is taken that there are operations of superposition other than addition which have practical interest. In these circumstances a system can be said to obey a law of generalized superposition if in response to $\alpha \circ \beta$ it yields $\alpha' \square \beta'$ where the rules of combination, $\circ$ and $\square$, need only have the same algebraic properties as addition.

If a system of interest obeys a law of superposition, tremendous benefits ensue, because it is possible to analyze the responses of that system by superposing known responses to a set of standard excitations from which any excitation may be formed. Of equal importance is the fact that all systems obeying a law of superposition can, as we shall see, be considered as a transformation of an ordinary linear system. This implies that they too can be used in such filtering tasks as signal separation, noise rejection, equalization, and waveshaping.

A direct consequence of the fact that the operations of superposition, $\circ$ and $\square$, must have the same algebraic properties as addition is that any system characterized by these operations can be visualized as an ordinary linear system preceded and followed by suitable signal transformations. This situation is depicted in Fig. 1 which shows the two equivalent configurations.

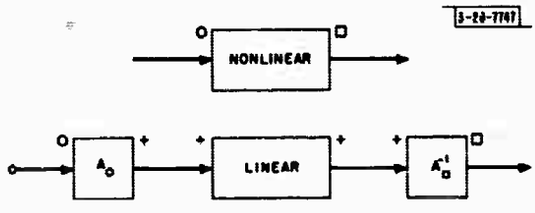

Fig. 1. Equivalent nonlinear configurations.

The signal transformations $A_{\bigcirc}$ and $A_{\square}^{-1}$ can be said to map the operation ○ into addition and addition into the operation □, respectively. For any particular pair of operations ○ and □, which incidentally may be identical, the whole class of nonlinear systems is defined and its one-to-one correspondence with the class of linear systems is placed in evidence.

2. Multiplicative Superposition and Image Processing

Image formation is fundamentally a multiplicative process. This statement applies equally to natural and photographic images. In a natural scene the illumination and reflectance of an object are combined by multiplication to form its observed brightness. The illumination and reflectance of a scene vary independently from object to object and from point to point forming a composite brightness image which in terms of its projection onto the retina is a two-dimensional spatial signal. If we wish to process images using a system which combines its signals according to the law of image formation, it is obvious that this system must obey superposition multiplicatively. Such a system is depicted in Fig. 2 in a manner analogous with Fig. 1. Observe

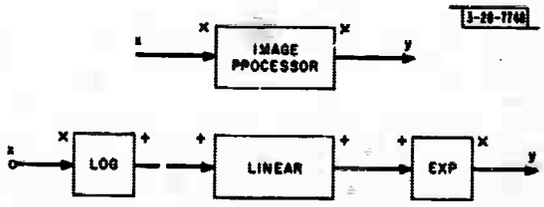

Fig. 2. Example of multiplicative superposition.

that the processes which map multiplication into addition and vice versa are specified explicitly and are in fact the familiar logarithm and exponential functions, as might be expected. The remaining nature of the image processor depends only on the linear system in the middle of the diagram.

Suppose that one desired to use this image processor as a filter to separate the components of an image, specifically the illumination and reflectance components. Since, generally speaking, illumination varies slowly from point to point whereas reflectance varies rapidly especially at object boundaries, a first approximation toward pulling these two components apart is to use a low-pass filter to obtain the illumination component and a high-pass filter to obtain the reflectance component. In the event that the Fourier components of the log illumination and the log reflectance occupy overlapping sections of the frequency spectrum, only partial separation of these components is possible, and the theory of optimum filtering can be brought to bear to effect a best separation.

If the linear component of the image processor is chosen as a simple amplifier (or attenuator) with gain K, the image processor becomes a power law device. The output, y, is then given in terms of the input, x, by the equation

$$y = x^K \quad . \tag{1}$$

Photographers can control the value of K, which they call γ, by selecting from a variety of photographic materials and using shorter or longer development times. For a subject, x, with a large light-to-dark ratio, K may be selected as less than unity so that the image, y, has a

reduced ratio and may be printed without exceeding the range of the photographic medium. The subjective effect is a "muddy" or "washed out" appearance if K is chosen too small, so some loss of high light and low light detail must be tolerated for scenes of very high contrast.

Since the linear component of the image processor can have different gains for different frequencies, it is possible to control photographic γ independently at each frequency. If one desires to maintain a large contrast for the details of an image but is willing to reduce contrast of large-area brightness variations, the gain of the linear component should be unity for large frequencies and less than unity at low frequencies. In this way one might avoid producing images of dull appearance while still producing a reduced overall light-to-dark ratio. This statement has been supported by tests involving a computer simulation of the image processor.

During these tests an image with an extreme contrast ratio was scanned into the TX-2 computer memory, its logarithm was computed, suitable two-dimensional filtering was performed, and the results exponentiated. When photographed from a cathode-ray tube display, the images could be viewed normally. The results were that dark areas of the original picture could be made far more visible as if illuminated with auxiliary lights without disturbing the rendition of the brightly lighted areas by using a value of $K = 1/2$ for low spatial frequencies and a value of $K = 1$ for high spatial frequencies. Additionally, if a value of $K > 1$ (e.g., $K = 2$) were used at high frequencies while maintaining $K = 1/2$ at low frequencies, the same brightening of dark areas was possible in cooperation with the sharpening of subjective detail in both light and dark areas.

These results are better understood by again considering the subject as a product of a slowly varying illumination function and a rapidly varying reflectance function as in Eq. (2):

$$x = i \times r \quad . \tag{2}$$

When processed the image produced is given by combining Eqs. (1) and (2) to produce Eq. (3):

$$y = i^{K_{low}} \times r^{K_{high}} \tag{3}$$

where K_{low} and K_{high} are the low and high frequency gains respectively. Since large brightness ratios in a subject are usually produced by large variations in illumination, the fact that y contains the square root of the original illumination explains why the brightness ratio is reduced. On the other hand, the reflectance component has the same variability or greater variability than in the original and thus details are preserved or enhanced.

The use of these methods in the processing of images has implications whenever dynamic range is limited and the preservation of details is important. It has the additional advantage of obeying a law of superposition which facilitates analysis through classical techniques while operating according to the same rules of combination that form the original subject information.

Some recent approaches to the problem of television bandwidth reduction[9] center around the idea of separating images into a relatively narrow bandwidth low frequency component and the complementary high frequency component, and preserving only a small fraction of the information in the high frequency component in the form of edge contours. At the receiver the edge contours are used in an attempt to recreate the high frequency component, and the results are combined with the low frequency component to recreate an approximation to the original image.

In these approaches the low and high frequency components are taken in the additive sense in keeping with the long established tradition of signal processing. There is reason to believe that if these components were taken in the multiplicative sense the results might be considerably enhanced. The primary motivation for this belief is the fact that the process of forming edge contours is basically a process of threshold detection. When the components are taken in the additive sense, poorly illuminated edges may be undetected when the threshold is set high enough to avoid spurious detection. In the multiplicative case all edges are represented equally by the high frequency component, since variations in illumination have been separated out in the low frequency component.

It has been known for a long time that the human eye is sensitive to the logarithm of light intensity rather than to the intensity itself. This statement is most strongly supported by Weber's law which demonstrates that for two adjacent uniformly illuminated areas with brightnesses B_1 and B_2, it is the brightness ratio threshold $(B_1 - B_2)/B_1$ which remains constant over an extremely wide range of brightnesses not the brightness difference $B - B_2$. It has also been recognized that the eye is a high-pass filter which is more sensitive to changes in intensity from point to point than to absolute intensity over a large area. The resulting implication that the eye might obey a principle of superposition based on multiplication is a most interesting one. If true, it might point the way to a quantitative tool for studying part of the seeing process.

3. Progress and Results

The project involving the nonlinear processing of photographic images has four basic phases:

(1) Fundamental digital photography
(2) Simultaneous contrast enhancement and dynamic range reduction
(3) Bandwidth reduction
(4) Psychophysical experimentation.

The results are summarized by phase.

Phase A:— A scanning algorithm was programmed to permit a new standard of storing scanned pictures. This was necessary to allow the retention of low illumination information contained by 12-bit linearly quantized samples. The need for such a large number of quantum levels is imposed by the fact that images are intended for dynamic range compression during which even the most poorly illuminated areas are substantially brightened. Memory constraints dictate that picture samples be limited to nine bits for final storage; however, the use of 9-bit linear quantization caused noise to become visible after compression. Under the new standard the logarithm of the 12-bit samples is stored, said logarithms rounded to nine bits.

Original pictures are 4 × 5 negatives printed on low contrast paper next to a calibrated grayscale. The original tone range of the negative can be reclaimed by the computer by referencing the grayscale.

The existing curve drawing display was programmed to serve as an output scanner. All scanned pictures are 340 × 340 samples in size. Each picture line is scanned using a fixed beam current and a variable velocity of scan to achieve intensity modulation. Each picture line is

repeated interlaced three times when the visibility of scan lines is to be minimized. Scan time is about thirty seconds per image.

Polaroid type 42 film and type 46-L transparencies are used throughout. The nonlinear density-exposure characteristics of these photographic materials are digitally compensated at display time. A grayscale accuracy of 10 percent or better is maintained throughout the usable range of each material. Output photographs give an impression of unusually high quality.

Phase B:– A flexible program for applying multiplicative filtering to images was completed and many pictures have been processed using it. The facility allows independent and precise contrast control of large area and small area brightness variations. By employing reduced large area contrast and normal small area contrast it is possible to reduce the overall dynamic range of an image so that it can be rendered using materials or equipment of limited range without severe distortion. By employing increased small area contrast coupled with decreased large area contrast the visibility of details can be enhanced without increasing the dynamic range of an image.

These results are achieved by subjecting images to linear spatial filtering while they are in the logged state. The frequency responses used are somewhat nominal in character, a broad class of characteristics producing similar results to an uncritical observer. The effect of small differences upon the critical observer is being studied.

Phase C:– The well-known image bandwidth compression method employing "artificial highs" was evaluated as applied to the logarithm of images. Advantages that were hypothesized on the basis of the insensitivity of the high frequencies of a logged image to variations in scene brightness are being studied. Initial results are extremely promising. Bandwidth reduction factors that are usually achieved using these methods remain available. At the same time, large errors in low brightness areas are avoided.

Phase D:– An hypothesis concerning that part of the human vision mechanism which can be usefully modeled as a multiplicative image processor is being studied. Attempts to nullify a set of well-known visual illusions by means of a single multiplicative filter are being planned.

E. Circuit Design Layout

Computer assistance in the design of integrated circuit masks is a natural application for the graphics capabilities of the TX-2. Besides providing an evaluation of available graphics techniques, the problems of integrated circuit design are of interest to our machine design group. Therefore, mask design programs have been created during the past few months with the following objectives:

(1) Provide a service for the rapid design and production of usable masks,

(2) Evaluate the graphics capabilities developed on the TX-2,

(3) Learn about the process of creating and programming a computer-aided design system.

The third subject merits some initial discussion.

A computer-aided design system starts with a user's requirements for assistance. Generally, it has been our experience that these requirements are insufficiently specified for

programming purposes. Most users do not know, or cannot say, explicitly what they need. We have come to believe that useful graphics systems must evolve in response to practical on-line experience with successive systems. The user himself must play a central role in providing feedback to the system designer who must be capable of responding rapidly to requests for modification. The programming tools that allow for rapid and efficient modification of programming systems are an absolute requirement. Given appropriate programming tools and hardware environment, the graphics system programmer still has a difficult job. He must devise a useful synthesis between available input and output techniques and the peculiarities of the problem being considered. Designing an input and display language for man-machine dialogue that will be convenient, unambiguous, and effective for a nonprogramming user is a good deal more difficult than its actual implementation.

In the development of our mask layout programs, we first programmed a system with arbitrary decisions made by the system programmer on the basis of sketchy user requirements. The input technique initially chosen was that of simple on-line pen motion recognition rather than the use of light buttons or function keys. The first system including the recognizer was written entirely in LEAP and required about 100 man hours of effort. A film of this program's operation is available.* A user first sketches in a circuit diagram then changes over to a mask configuration and can rearrange the components for a satisfactory layout.

This program was not intended to produce masks but rather to provide the mask designers and the system programmers with a common frame of reference. Since then progress has been rapid. With approximately 50 additional man hours of programming effort, the system progressed through two more stages and now can produce mask sets as shown in Fig. 3.

The man-hour expenditures indicate the power of LEAP as a programming tool. Constructs are available in LEAP for utilizing the display and file management capabilities of APEX, and the LEAP data structuring facilities have been most convenient. Much of the tedious work involved in creating displays, defining structures, keeping files, etc., has been done once by APEX or LEAP and made easily available for any application. Adding a library facility to the mask design program for naming, storing, and retrieving individual circuit definitions was literally a 30-minute, on-line programming task. All the difficult work had already been done; the high level definition of the library actions required only a few statements.

This application program has also provided feedback to the language designers. At present LEAP can store and retrieve individual data structures (circuits in this application), but cannot conveniently merge them as desired by the circuit designer. This capability will be incorporated in LEAP for general use at a future date. Our approach has been to augment the language and its compiler with appropriate general purpose constructs to meet new requirements. The VITAL system makes such changes possible with relative ease. As another example, LEAP now contains special variables which simplify input tablet programming.

The mask design program at present has the computer doing only a small part of the job. The computer serves as a drafting assistant and allows the designer to draw, erase, and move mask components. It also pays attention to spacing tolerances in a very simple way. The pen-motion input has been augmented with light buttons and keyboard input. A better pen-motion

* "Circuit Sketching on the TX-2 Computer."

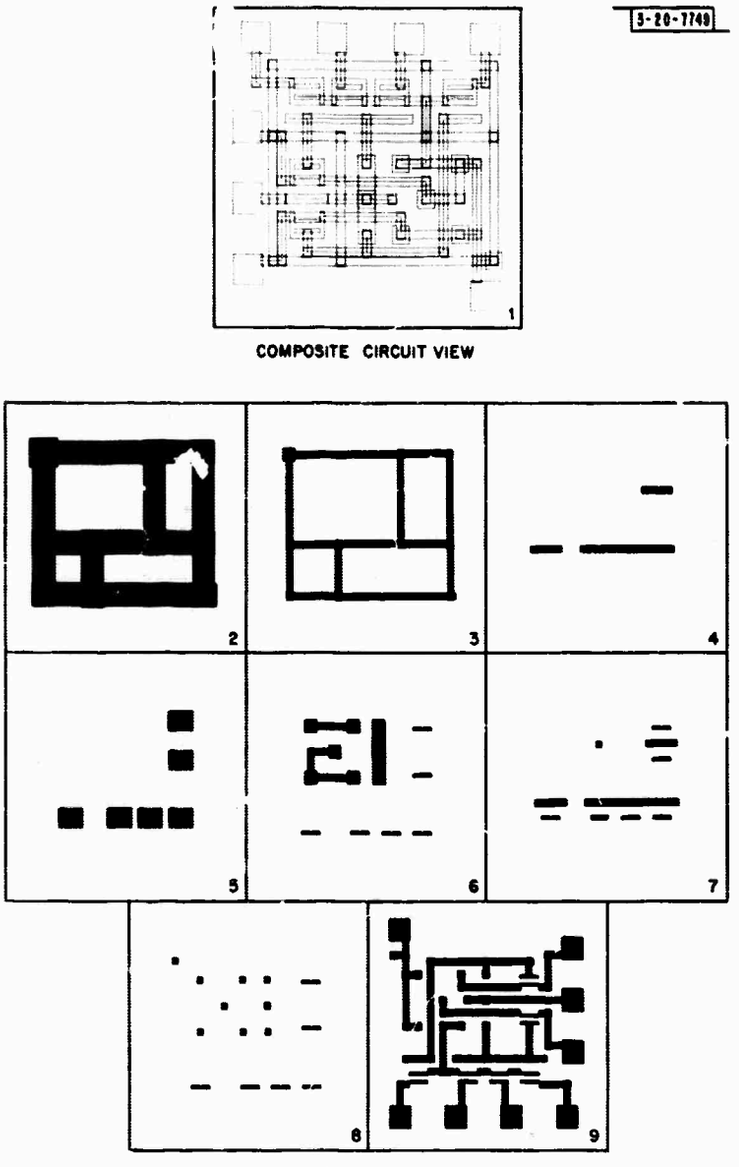

Fig. 3. Compu'er-generated integrated circuit masks.

recognizer is planned and will reduce the need for these other kinds of input. The program does not do automatic placement or wire routing and it does not check the man's final design against a circuit standard. All these and other additional features are under consideration. The difficult decision for each is to decide whether it is really useful, and if so, how the user at the console will control the additional planned computer capability. Does he type, poke a button, or draw new symbols? If he can say something to the machine, how does he change his mind and unsay it? Our experience has shown that the actual implementation is not difficult once these questions are answered.

We hope in the near future to fabricate trial circuits from computer-generated masks in order to provide the ultimate feedback necessary to make a useful operational facility.

REFERENCES

1. Thomas G. Stockham, Jr., "Some Methods of Graphical Debugging," Proceedings of the IBM Scientific Computing Symposium on Man-Machine Communications, 3 – 5 May 1965, pp. 57 – 71.
2. Thomas G. Evans and Lucille D. Darley, "On-Line Debugging Techniques: A Survey," Proceedings of the Fall Joint Computer Conference, 1966, 29, pp. 37 – 50.
3. L.G. Roberts, "Graphical Communication and Control Languages," Proceedings – Second Congress on Information System Sciences, Hot Springs, Virginia, 1964, pp. 211 – 217.
4. W.R. Sutherland, "On-Line Graphical Specification of Procedures," Technical Report 405, Lincoln Laboratory M.I.T. (23 May 1966) DDC 639734.
5. See Ivan E. Sutherland, "Sketchpad: A Man-Machine Graphical Communication System," Technical Report 296, Lincoln Laboratory, M.I.T. (30 January 1963), DDC 404549, followed by Refs. 3 and 4. Details of the display file organization are found only in several Lincoln Laboratory internal memos.
6. J.A. Feldman, "Aspects of Associative Processing," Technical Note 1965-13, Lincoln Laboratory, M.I.T. (21 April 1965) DDC 614634; P. Rovner and J.A. Feldman, "An Associative Processing System for Conventional Digital Computers," Technical Note 1967-19, Lincoln Laboratory, M.I.T. (21 April 1967), DDC 655810.
7. L.F. Mondshein, private communication.
8. A.V. Oppenheim, "Superposition in a Class of Nonlinear Systems," 1964 IEEE International Convention Record, Part 1, pp. 171 – 177.
9. D.N. Graham, "Image Transmission by Two-Dimensional Contour Coding," Proc. IEEE 55, No. 3, pp. 336 – 346 (March 1967).

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Lincoln Laboratory, M.I.T.	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP None

3. REPORT TITLE

Semiannual Technical Summary Report to the Advanced Research Projects Agency for Graphics

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Semiannual Technical Summary Report (1 June through 30 November 1967)

5. AUTHOR(S) (Last name, first name, initial)

Raffel, Jack I.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
30 November 1967	36	9

8a. CONTRACT OR GRANT NO. AF 19(628)-5167 b. PROJECT NO. ARPA Order 691 c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) Semiannual Technical Summary for 30 November 1967 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) ESD-TR-67-570

10. AVAILABILITY/LIMITATION NOTICES

This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
None	Advanced Research Projects Agency, Department of Defense

13. ABSTRACT

The APEX display executive has been in regular operation and experience with new display hardware and a remote display console has been helpful in pinpointing needed changes. The APEX interrupt executive has been completed and will be evaluated during the next quarter. A new APEX scheduling algorithm has been implemented in order to improve the system response to users. The 338 remote console is operational in Washington, D.C., and is being used on a routine basis.

The compiler-compiler system VITAL is in regular use. The programming language LEAP, implemented using VITAL, has been used for programming constraint problems, a portion of the new VITAL system, integrated circuit mask layout, and two debugging packages. These latter present their data graphically to the user. The design of an improved VITAL system is under way.

The earth display program created earlier in the year has been used on actual data obtained from LES-4. A new approach for solving constrained systems is under development which uses a geometrical model. Several trial applications of this method have been programmed using LEAP. A special compiler and language for testing integrated circuits have been developed.

A theory of generalized superposition has been applied to waveform processing. Programs have been developed for scanning, processing, and photographing pictures which are subjected to this new kind of filtering. Programs for integrated circuit mask layout are under development which will allow designers to sketch in mask components and rearrange them in search for a satisfactory layout. Photographs can then be taken of the resulting masks.

14. KEY WORDS

graphical communication
TX-2
time sharing
man-machine
display systems